Fast XML Structural Join Algorithms by Partitioning

Nan Tang Jeffrey Xu Yu Kam-Fai Wong
The Chinese University of Hong Kong, Hong Kong, China
Jianxin Li
Swinburne University of Technology
email: {ntang,yu,kfwong}@se.cuhk.edu.hk jili@ict.swin.edu.au

Abstract

An XML structural join evaluates structural relationship (parent-child or ancestor-descendant) between XML elements. It serves as an important computation unit in XML pattern matching. Several classical structural join algorithms have been proposed such as *Stack-tree* join and *XR-Tree* join. In this paper, we consider to answer the problem of structural join by partitioning. The Dietz numbering scheme is used for encoding since nodes with the Dietz encodes could be well distributed in a plane. We first extend the relationships between nodes to the relationships between partitions in a plane and get some observations and properties about the relationships between partitions. We then propose a new partition-based method *P-Join* for structural join between ancestor and descendant nodes based on the properties derived from our observations. Moreover we present an enhanced partitioned-based structural join algorithm and two optimized methods. Extensive experiments show that the performance of our proposed algorithms outperform that of *Stack-tree* and *XR-Tree* algorithms. In order to store the partitioning results, we design a simple but efficient index structure *PSS-tree*. The experimental result shows that it has less maintenance overhead than *XR-Tree*.

Keywords: XML, Structural Join, Partition
ACS Classification: H.2 (Database Management)

1 Introduction

XML has been increasingly used in Web applications and become the *de facto* standard for data exchange over the Internet. Queries in XML query languages (see, e.g., [4,9,11,14]) typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. These queries can be naturally decomposed into a set of basic parent-child ($/$) and ancestor-descendant ($//$) relationships between pairs of nodes, called structural joins.

Structural join is known as an important computation primitive in XML query processing. It is not only the basic unit of the cost-based twig pattern matching [29] but also the basis of holistic twig join algorithms [5, 21]. Therefore, improving the performance of structural joins remains to be an important research issue.

There exists a rich body of work on structural joins. For example, [31] proposed a variant of the traditional merge join algorithm, called multi-predicate merge join (MPMGJN). However, it may perform a lot of unnecessary computation and disk I/O for matching structural relationship. Similarly, $\mathcal{EE}$-Join and $\mathcal{EA}$-Join in [22] may scan an element set multiple times. The Stack-Tree-Desc join algorithm proposed in [2] improved the merge based structural join algorithms with a stack mechanism. Only one sequential scan is needed for two input ordered lists, *AList* and *DList*.

Index-based algorithms push the performance of structural joins to a next level [12, 20]. The essential idea is to use indexes on the participating element sets to directly (or near directly) find the matching elements and skip those without matches. Experiments show that index-based algorithms could be faster than merge-based algorithms by orders of magnitude.

Besides index-based algorithms, partitioning is another way to accelerate structural joins. [23] proposed three partition-based path join algorithms, namely descendant partition join, Segment-tree partition join and ancestor link partition join. These partition-based join algorithms could make better use of the buffer memory by using small in-memory data structures and taking advantage of unevenly sized inputs, but they did not utilize the positional characters of the encodings of tree nodes well.

Furthermore, many numbering schemes have been proposed because with them we could quickly determine the positional relationships between tree nodes. Previous strategies solve the problem of structural joins by using the basic properties of numbering schemes. Specifically, they mainly pay attention to the relationship of encoding nodes while ignoring the spatial distribution of tree nodes.

In this paper, we first introduce a partition-based structural join method *P-Join* [26] specially designed for XML queries by extending the relationship between nodes to the relationship between partitions. Then we present an enhanced partition-based structural join algorithm and two optimized algorithms to further improve the performance of *P-Join*.

The major contributions of this paper are summarized as follows:

- We extend the relationships between nodes to the relationships between partitions and get some observations and properties about the relationships between partitions. Based on the relationships between partitions, we get 9 observations and 4 properties.
- We present a partition-based structural join method, namely, *P-Join*. Previous work [12, 21] only utilized *B^+-Tree* and *XR-Tree* on *Stack-tree* join algorithm to skip ancestors and descendants nodes. With *P-Join*, however, after filter operation, the ancestor and descendant nodes are further classified into different areas using positional relationships in a plane. Then some portions could directly output join results without actual join operation, other portions could also efficiently produce join results utilizing spatial characters.
- We present an enhanced partition-based structural join algorithm and two optimized methods based on *P-Join*, and we also design an efficient structure *PSS-tree* to index partitioning results.
- We report the result of a performance study which shows that our proposed algorithms outperform previous classical structural join algorithms.

The rest of this paper is organized as follows. In Section 2 we introduce the numbering scheme selected and some concepts about partition relationships. Next, in Section 3 we present a new partition-based structural join method *P-Join*. Section 4 presents an enhanced partition-based structural join algorithm and two optimized methods based on *P-Join*. Section 5 gives the performance evaluation. Finally Section 6 concludes this paper.

2 Numbering Scheme and Partition Relationships

XML document is commonly modeled as a data tree. Nodes in the tree correspond to elements, attributes or values, and the edges represent immediate parent-child relationships. A unique feature of XML queries is to query the positional relationships between tree nodes.

Efficiently determining the structural relationships between tree nodes is the pivot of XML queries. In this section we first introduce the numbering scheme adopted in this paper to judge the positional relationships of tree nodes, then we describe the spatial positional characters of points and partitions.

There are many proposed numbering schemes for efficiently determining the positional relationships between any pair of tree nodes [2, 13, 16, 22, 24, 28]. In this paper we attempt to make use of a numbering scheme for partitioning data that will in turn make the structural join more efficient. Therefore, the choice of a good numbering scheme is important. In this paper, we use the Dietz numbering scheme [13], which uses *preorder* and *postorder* values, as a pair, to encode each tree node, where *preorder* is the order in pre-order traversal of XML data tree, while *postorder* is the order in post-order traversal.

Dietz numbering scheme expresses the positional relationship as follows:

- For a tree node e_j and its ancestor e_i, $\text{PRE}\,(e_i) < \text{PRE}\,(e_j)$ and $\text{POST}\,(e_i) > \text{POST}\,(e_j)$.
- For two sibling nodes e_i and e_j, if e_i is the predecessor of e_j in pre-order traversal, $\text{PRE}\,(e_i) < \text{PRE}\,(e_j)$ and $\text{POST}\,(e_i) < \text{POST}\,(e_j)$.

Here, PRE (e) and POST (e) represent the $preorder$ and $postorder$ of element e, respectively.

Figure 1 shows a simple XML data tree encoded with the Dietz numbering scheme.

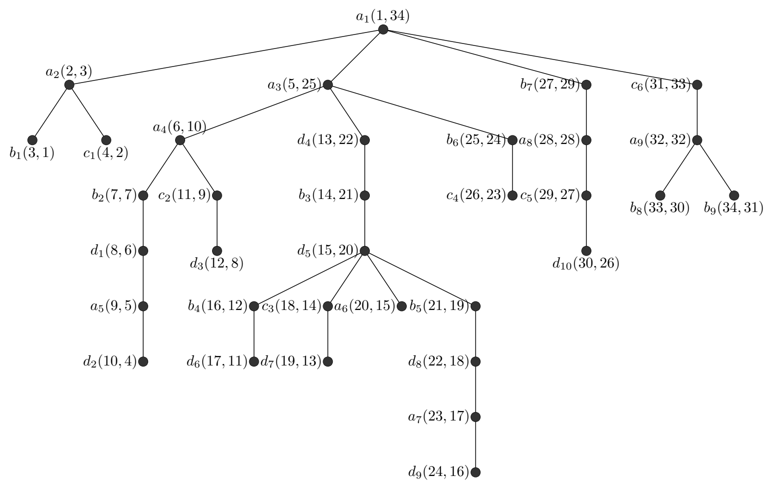


Figure 1: An XML document example

With the Dietz numbering scheme, the structural relationship between tree nodes can be clearly expressed in a plane as shown in Figure 2. The x-axis represents the *preorder* values and the y-axis represents the *postorder* values. For any XML data tree, no two XML nodes have the same *preorder* or *postorder* values. Therefore, no two XML nodes can be placed at the same row or column on a plane. Consider the node p in Figure 2. Any node e on the top left of p is the ancestor of p because $\text{PRE}\,(e) < \text{PRE}\,(p)$ and $\text{POST}\,(e) > \text{POST}\,(p)$; likewise, the node on the bottom right is the descendant of p; the node on the bottom left is the preceding node of p; the node on the top right is the following node of p.

The main reason of using the Dietz numbering scheme is simply explained below. In comparison with the region code, which uses *(start, end)* pair for coding [2]. Consider the XML document example in Figure 1. Figure 3 (a) shows the distribution of nodes in a plane under Dietz numbering scheme. Figure 3 (b) shows the distribution of nodes using region numbering scheme. For simplicity, in Figure 3, we only show nodes a and d while omitting b and c from the example XML document shown in Figure 1. As can be seen in Figure 3, nodes with the Dietz numbering scheme are distributed dispersedly, whereas the nodes with the region numbering scheme are rather skewed and only in the upper triangle of a subspace. This distribution character of Dietz numbering scheme is more appropriate for our spatial-partitioning technique introduced in section 4, thus we choose Dietz numbering scheme.

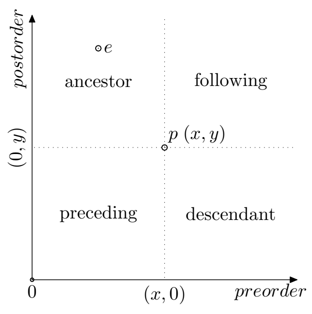


Figure 2: Project Tree Nodes

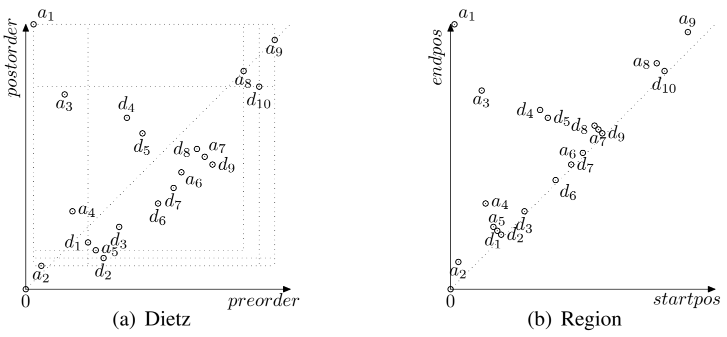


Figure 3: Dietz and Region

In this paper, we extend the relationships between nodes as shown in Figure 2 to the relationships between partitions. We show an example in Figure 4, where the whole space is partitioned into 9 disjoint rectangle partitions, $R_1, R_2, \cdots, R_9$. Here each R_i contains a subset of XML elements (or nodes in an XML tree). Consider the partition R_5 in the center of Figure 4. We made the following observations:

1. All nodes in R_1 are the *ancestors* of all nodes in R_5.
2. All nodes in R_3 are the *following* nodes of all nodes in R_5.
3. All nodes in R_7 are the *preceding* nodes of all nodes in R_5.
4. All nodes in R_9 are the *descendants* of all nodes in R_5.
5. Some nodes in R_2 are the *ancestors* or *following* nodes of some nodes in R_5.
6. Some nodes in R_4 are the *ancestors* or the *preceding* nodes of some nodes in R_5.
7. Some nodes in R_6 are the *descendants* or *following* nodes of some nodes in R_5.
8. Some nodes in R_8 are the *descendants* or *preceding* nodes of some nodes in R_5.
9. Some nodes in R_5 may have any positional relationships with nodes in R_5.

Note, items 1-4 state that a relationship that is always true for all nodes in two different partitions. Items 5-8 state that a relationship that may be true for some nodes in two different partitions. The last item 9, states a fact a partition-based positional relationship does not deal with the relationships between nodes inside a single partition.

The observations made above show that we can process structural joins using a partition-based approach. When users query the ancestors of nodes in the partition of R_5, (1) we do not need to do structural join between nodes in R_5 and any node in R_3, R_6, R_7, R_8 and R_9, since none of these partitions includes ancestors of any nodes in R_5; (2) we do not need to do structural join between R_1 and R_5, since all nodes in R_1 are the ancestors of all nodes in R_5; and (3) we only need to do structural join between R_2, R_5 and R_4, R_5, since some nodes in R_2 and R_4 are the ancestors of some nodes in R_5. The similar techniques can be applied when users query the descendant or other XPath axes in the partition of R_5.

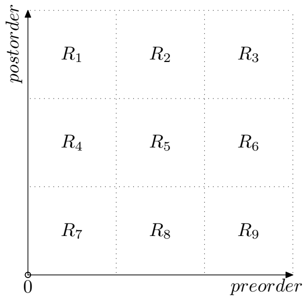


Figure 4: Partitions

Based on the above observations, we define two predicates namely, $all(S, R_i, R_j)$ and $some(S, R_i, R_j)$ where R_i and R_j are partitions and S is a type indicator including `ancestor`, `descendant`, `following`, and `preceding`. Here, $all(S,\ R_i, R_j)$ means that all nodes in R_i satisfy the given relationship S with R_j, and $some(S, R_i, R_j)$ means some nodes in R_i satisfy the given relationship S with R_j. For example, as shown in Figure 4, $all(ancestor, R_1, R_5)$ is true, $all(ancestor, R_5,\ R_1)$ is false, and $all(ancestor, R_4, R_5)$ is false. It is worth noting that, given a partition R_i, only those partitions that are at top left of R_j satisfy $all(ancestor, R_j, R_i)$, and only those partitions that are at bottom right of R_j satisfy $all(descendant, R_j, R_i)$.

Given a partition R_i, there exist two kinds of partitions (R_j) that satisfy $some(ancestor, R_j, R_i)$, which are the partitions that are at *up* position of R_i and the partitions at *left* position of R_i. For example, $some(ancestor, R_2, R_5)$ is true, since R_2 is up R_5, denoted $up(R_2, R_5)$; $some(ancestor, R_4, R_5)$ is true, since R_4 is left to R_5, denoted $left(R_4, R_5)$. In a similar fashion, there exist two kinds of partitions (R_j) that satisfy $some(descendant, R_j, R_i)$, which are the partitions at *down* position of R_i and the partitions at *right* position of R_i.

PROPERTY 1. *Given two partitions R_i and R_j such as $up(R_j, R_i)$ is true. Assume that there are three nodes $x \in R_i$, $y \in R_j$ and $z \in R_j$. If y is an ancestor of x, then z is an ancestor of x if* $\text{PRE}\,(z) < \text{PRE}\,(y)$.

PROPERTY 2. *Given two partitions R_i and R_j such as $left(R_j, R_i)$ is true. Assume that there are three nodes $x \in R_i$, $y \in R_j$ and $z \in R_j$. If y is an ancestor of x, then z is an ancestor of x if* $\text{POST}\,(z) > \text{POST}\,(y)$.

PROPERTY 3. *Given two partitions R_i and R_j such as $down(R_j, R_i)$ is true. Assume that there are three nodes $x \in R_i$, $y \in R_j$ and $z \in R_j$. If y is a descendant of x, then z is a descendant of x if* $\text{PRE}\,(z) > \text{PRE}\,(y)$.

PROPERTY 4. *Given two partitions R_i and R_j such as $right(R_j, R_i)$ is true. Assume that there are three nodes $x \in R_i$, $y \in R_j$ and $z \in R_j$. If y is a descendant of x, then z is a descendant of x if* $\text{POST}\,(z) < \text{POST}\,(y)$.

These properties can be used to reduce the overhead of evaluating structural joins, when nodes are sorted in a partition. Note that [16,18] utilizes the Dietz numbering scheme, as an index structure to support XPath location steps including ancestor, following, preceding, sibling, descendant-or-self, etc. They make use of the encoded values to judge the positional relationship between *points* which only locates one node at one time. Here, in this work, we focus ourselves on positional relationship between *partitions*, which locates a set of nodes at one time.

3 A New Partition-Based Structural Join

In this section, we propose a new partition-based method for structural join between ancestors and descendants. Our method can be easily extended to support other XPath axes including following, preceding and

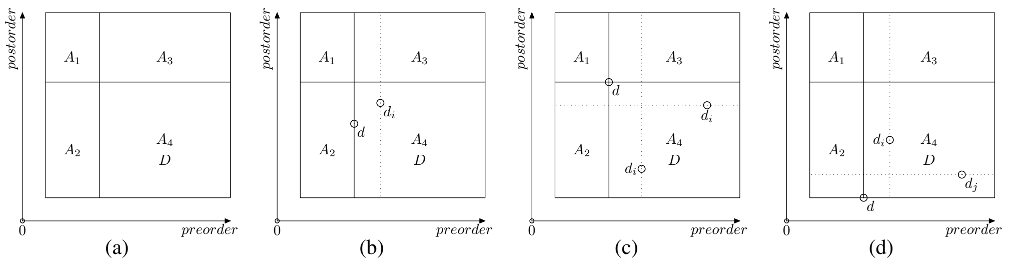


Figure 5: Several Cases

sibling, etc.

First, we show several cases in Figure 5, and then we outline our partition-based structural join algorithm. Figure 5 (a) shows a general case where a plane is partitioned into 4 partitions, A_1,A_2, A_3 and A_4 according to D. Here, D is a minimal partition (rectangle) that contains a set of descendants. D is always located the same as A_4 for any structural join based on ancestor/descendant relationships.

One technique we use in our partition-based join is to dynamically adjust the partitions of A_1, A_2, A_3 and A_4 $(= D)$ when we attempt to find ancestors for every $d \in D$. In other words, the four partitions are conceptually redrawn for any $d \in D$. Two ascending *B*$^+$*-Tree* indices on the $preorder$ and $postorder$ value are created for A, respectively. Note that we only need to scan every $d \in D$ once. The details are given below. We assume that elements in partitions are sorted by their $preorder$ values in ascending order. The adjustment of partitions has the following three cases for a descendant $d \in D$.

1. d is in the middle of the boundary between A_2 and D (Figure 5 (b)).
2. d is at the top of the boundary between A_2 and D (the top-left corner of D) (Figure 5 (c)).
3. d is at the bottom of the boundary between A_2 and D (Figure 5 (d)).

For the first case, we output d with all a in A_1, then traverse each a_i in A_2 in the reverse order, and compare it with d, if a_i is not an ancestor of d, then all elements d_i after d cannot be the descendant of a_i, we do not need to join this a_i with any such $d_i \in D$. Otherwise, if a_i is an ancestor of d, all a_j before a_i are ancestors of d according to Property 2. Suppose that after processing d, we need to process the next descendant d_i in the sorted list (Figure 5 (b)). Because d is on the boundary between A_2 and D, the boundary between A_3 and $d_i \in D$ will remain unchanged. We can find those elements $a \in A_3$ such as $\textsc{Pre}(a) < \textsc{Pre}(d_i)$. Note that when processing $d_i \in D$ in Figure 5 (b), those nodes a in A_4 such as $\textsc{Pre}(a) < \textsc{Pre}(d_i)$ are adjusted to A_2. It is important to note that the area of A_1 and A_2 are expanded while the area of A_3 and A_4 are shrunk. This adjustment process is shown in Figure 5 (b).

The second case is simple, since only all nodes in A_1 are the ancestors of d. All nodes in A_2 are the preceding nodes of d, all nodes in A_3 are the following nodes of d, and all nodes in A_4 are the descendants of d. We can simply output d with every $a \in A_1$ as join results. After processing d, we need to adjust the boundaries of D, when we process the next descendant d_i which has the smallest $preorder$ value. In the first step, we identify a node $d_j \in D$ with the largest $postorder$ value from the unprocessed descendants. In the second step, we adjust those nodes the following nodes to the A_1.

- $a \in A_2$ if $\textsc{Pre}(a) > \textsc{Post}(d_j)$,
- $a \in A_4$ if $\textsc{Pre}(a) > \textsc{Post}(d_j)$,
- $a \in A_3$ if $\textsc{Pre}(a) < \textsc{Pre}(d_i)$, and
- $a \in A_4$ if $\textsc{Pre}(a) < \textsc{Pre}(d_i)$.

In the third step, we adjust those nodes a in A_4 such as $\textsc{Post}(a) > \textsc{Post}(d_j)$ to A_3. This adjustment process is shown in Figure 5 (c).

For the third case, all nodes in A_1 and A_2 are ancestors of d, and no nodes in A_3 and A_4 are ancestors of d. After processing d, we get the next element d_i in D following the sorted order. We can determine a node d_j which has the minimum $postorder$ value from the left elements. Because d is at the bottom of the boundary between A_2 and D, the boundaries need to be adjusted. Here, we first remove those the elements

a in A_2 and A_4 if Post (a) < Post (d_j), since these nodes a will not be ancestors of any descendants $d \in D$. Second, we adjust those elements a in A_3 if Pre (a) < Pre (d_i) to A_1. Third, we adjust those elements $a \in A_4$ if Pre (a) < Pre (d_i) to A_2. This adjustment process is shown in Figure 5 (d).

The partition-based structural join algorithm, P-Join(A, D), is outlined below.

Algorithm 1 P-Join (A, D)

```
1:  $A_1, A_2, A_3, A_4 \leftarrow$ Partition$(A, D)$
2:  for each $d \in D$ (sorted) do
3:      if $d$ is in case 1 then
4:          $posFlag \leftarrow$ MIDDLE
5:      else if $d$ is in case 2 then
6:          $posFlag \leftarrow$ TOP
7:      else if $d$ is in case 3 then
8:          $posFlag \leftarrow$ BOTTOM
9:      end if
10:     Output $(A_1, d)$
11:     if $posFlag =$ MIDDLE then
12:         Locate $(A_2, d)$; Output $(A_2, d)$; Adjustmiddle $(A, d)$
13:     else if $posFlag =$ TOP then
14:         AdjustTopLeft $(A, D)$
15:     else if $posFlag =$ BOTTOM then
16:         Output $(A_2, D)$; AdjustBottomLeft $(A, D)$
17:     end if
18: end for
```

The main advantage of algorithm *P-Join* is that it does not necessarily join every ancestors and descendants using a dynamic boundary adjustment technique.

In fact, as a pre-processing technique, we can filter those nodes that cannot be matched before calling *P-Join*. Filtering operation is extensively used in database system to save I/O cost, consequently improve the system performance. Traditional bit filter [3, 27] techniques in relational database utilize the equal relationship between attributes of two relations for filtering. For semistructured data like XML, however, bit filter techniques cannot be applied well, thus we filter nodes with their spatial positional characters.

We explain filtering using Figure 3 (a). First, we use the minimum rectangle containing all descendants d to filter A, all ancestors a in the area that cannot be an ancestor of any d are filtered. Thus, a_2 and a_9 are filtered, where a_2 is the preceding node of all nodes d and a_9 is the following node of all nodes d. We can remove a_2 and a_9 from A, and then we use the minimum rectangle containing the left A to filter D, all nodes d in the area that will not be a descendant of any a will be filtered. In this example, no d is filtered because any d will produce the structural join result of $a/\!/d$. It is easy to see that the filtering order of A and D is arbitrary and the results are the same. Moreover, one mutually filtering of A and D is enough and there is no need to recursively filter.

4 Optimization Techniques

In this section, we further investigate optimization techniques on how to partition A and D when querying the ancestor-descendant relationship $a/\!/d$. We will present a spatial partitioning approach and two optimization techniques, and will discuss the data structure for indexing the partitioning results. On top of the spatial partitioning approach, two new partition based structural join algorithms are proposed.

4.1 Spatial-Partitioning

Partitioning A and D for minimizing structural join is challenging, and is itself complex and time consuming. To find an efficient partitioning method, we are now left with the challenge to find a method which considers the spatial characters of A or D first, and then partition the leftover D or A for structural join.

When the size of D is large, the nested level is high and the distribution is poorly proportioned, the partitioning methods that mainly consider D will get good performance. On the contrary, the methods that pay attention to A will be better. The partitioning order of A or D is symmetrical. In this section, we mainly discuss the partitioning method for D, the method for A can be applied in a similar manner.

Logically, an element node should be potentially distributed in any area of a plane. In practice, however, for a general XML data tree, element nodes are mainly distributed near the area from the bottom left to top right of a plane. The root node will exist on the top left. Only when the whole XML data tree is a linear structure, the node will appear on the bottom right, which is obviously an abnormal case. Based on the characteristics of the general distribution of XML data trees, we propose an efficient spatial partitioning method, denoted as SPATIAL-PARTITIONING.

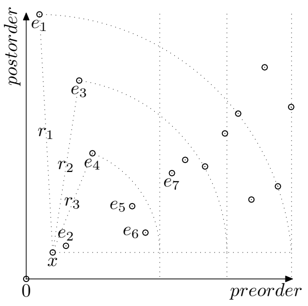


Figure 6: Search n Nearest Nodes

The basic idea of SPATIAL-PARTITIONING is first to find two data marks that have the largest distance in D and search the nearest $\frac{|D|}{N}$ nodes of each data mark with a NEAREST method. Spatial-Partitioning will recursively call the NEAREST method until the size of a group is 1 or 2. The two data marks are chosen from the edges of the minimum rectangle containing all d. The selection of the $\frac{|D|}{N}$ nearest nodes is implemented as shown in Figure 6. The NEAREST() procedure is given below.

Algorithm 2 **NEAREST** (x, n)

```
1:  $nX \leftarrow x$ 's nearest $n$ points in $x$-axis
2:  $e \leftarrow$ FURTHERMOST $(x, nX)$
3:  $r \leftarrow$ DISTANCE $(e, x)$
4:  $p \leftarrow minPreorder$ node in the left nodes
5:  while X-DISTANCE$(p, x) < r$ do
6:      if DISTANCE $(p, x) < r$ then
7:          $nX \leftarrow nX - e + p$
8:          $e \leftarrow$ FURTHERMOST$(x, nX)$
9:          $r \leftarrow$ DISTANCE$(e, x)$
10:         $p \leftarrow p.next$
11:     end if
12: end while
13: return $nX$
```

In the NEAREST procedure, the FURTHERMOST(x, nX) searches the furthermost point of x in nX, the DISTANCE(e, x) is the $2D$ distance between e and x, the X-DISTANCE(p, x) is their horizontal distance.

We give an example for searching five nearest nodes of x including x in Figure 6. We first search five nearest nodes on the X-axis distance of x, which are x, e_1, e_2, e_3 and e_4. The furthermost node is e_1, and r_1 is the distance between x and e_1, any node whose horizontal distance with x is less than r_1 may be in the result. We choose node according to its preorder, so next we get e_4, which is nearer than e_1, so we remove e_1 and add e_4. Now the five nodes are x, e_2, e_3, e_4 and e_5. And then, we get e_6 and remove e_3, now the furthermost node is e_4 and the furthermost distance is r_3, other nodes' horizontal distance with x are all larger than r_3. Now we get $x's$ five nearest nodes x, e_2, e_4, e_5 and e_6.

Compared with computing all nodes to find their nearest n nodes, this partitioning method only need to compute a small part of nodes to get the results. Furthermore, along with the progress of partitioning, the nodes will be less and the search speed will be quicker. The efficiency of NEAREST procedure benefits from the particular plane distribution characters of tree nodes. With above example, we can easily see that our partitioning method could quickly partition D into N groups, each has $\frac{|D|}{N}$ nodes. We show SPATIAL-PARTITIONING algorithm below.

Algorithm 3 SPATIAL-PARTITIONING $(D, N, i = 1)$

```
1:  if $N = 1$ then
2:      return $D_i$
3:  else if $N = 2$ then
4:      choose $p_1$ and $p_2$ in $D$
5:      $D_i \leftarrow$ NEAREST $(p_1, |D|/N)$
6:      $D_{i+1} \leftarrow$ NEAREST $(p_2, |D|/N)$
7:      return $D_i, D_{i+1}$
8:  else if $N > 2$ then
9:      choose $p_1$ and $p_2$ in $D$
10:     $D_i \leftarrow$ NEAREST$(p_1, |D|/N)$
11:     $D_{i+1} \leftarrow$ NEAREST$(p_2, |D|/N)$
12:     $D \leftarrow D - D_i - D_{i+1}$
13:     $N \leftarrow N - 2$
14:     $i \leftarrow i + 2$
15:     GROUP$(D, N, i)$
16:     return $D_1 \cdots D_N$
17: end if
```

In our SPATIAL-PARTITIONING() method, if $N = 1$, we just distribute all D to one group. Otherwise, we choose two enough remote datum marks on the edges of minimum rectangle containing all D, and search each datum mark's $\frac{|D|}{N}$ nearest points and distribute them to one group. And then, we recursively call the spatial partitioning method until all D are distributed to N groups. With this procedure, the number of d in each group is $\frac{|D|}{N}$.

To make the result simple and concrete, suppose partitioning number $N = 2$. We partition the D as shown in Figure 3 (a), we first choose two enough remote points d_1 and d_{10}. Then choose d_1's $\frac{|D|}{N} = 5$ nearest points d_1, d_2, d_3, d_5, d_6 to first group D_1 and $d_4, d_7, d_8, d_9, d_{10}$ to the second group D_2. The partitioning result is shown in Figure 7. D_1 is the minimum rectangle containing all d nodes inside D_1, and D_2 is the minimum rectangle containing all d nodes inside D_2.

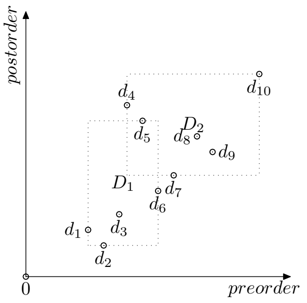


Figure 7: Partition D

4.2 Optimized Partitioning Methods

Under the precondition of average partition of D, however, we should further optimize the partitioning results to improve the performance. To get this requirement, we propose two optimized methods based on the partitioning results of SPATIAL-PARTITIONING() method.

The first optimized method is to minimize the maximum area of D. Minimizing the maximum rectangle of D could increase the possibility that sibling element nodes fall into the same rectangle, which could reduce the chance that the same ancestor a be duplicated in different groups. Thus save the additional disk I/O cost. The method MINMAXD($D_1, \cdots, D_N$) is given below:

Algorithm 4 MINMAXD $(D_1, \cdots, D_N)$

```
 1: $D_i \leftarrow$ MAXAREA($D_i$)($i = 1 \cdots N$)
 2: $D_j$ = NEARESTAREA(CENTER ($D_i$))
 3: $d$ = NEARESTPOINT ($D_i, D_j$)
 4: if AREA ($D_j + d$) < AREA($D_i$) then
 5:     MERGE ($D_j, d$)
 6:     CUT ($D_i, d$)
 7:     return MINMAXD ($D_1, \cdots, D_N$)
 8: else
 9:     return $D_1 \cdots D_N$
10: end if
```

This method first searches the rectangle area D_i from D_1 to D_N having the maximum area. Next we try to reduce the area of D_i. We search the area D_j whose center is the nearest to the center of D_i. Then we choose the point p on the edge of D_i, which could guarantee that after removing p from D_i, the area of D_i will reduce. Now we just need to judge if the area of new D_j with p is larger than the area of old D_i. We aim at minimizing the maximum area of D, therefore, we recursively call this method if this optimization could further reduce the maximum area. Otherwise, we terminate this method and return the partitioning result after optimization.

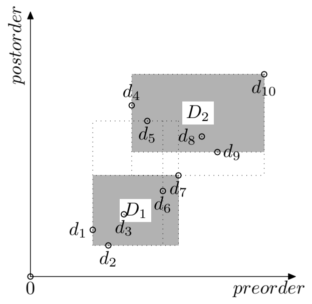


Figure 8: Minimize Maximum D

The second optimized method is to minimize each overlay area. Any element d in the overlay area will be distributed to just one group. Compared with the whole rectangle, the overlay area is a smaller area and elements in it are more likely to be the sibling elements. If sibling elements are distributed to distinct groups, their common ancestor are sure to be distributed to corresponding groups, which need redundant duplication overhead. Minimizing the overlay area of rectangles and the number of d in the overlay area may, therefore, reduce the entire duplication overhead. The optimized method MINOVERLAY($D_1, \cdots, D_N$) is given below:

This method first checks if there are overlay areas from D_1 to D_N. If there is not any overlay area from D_1 to D_N, no optimization is needed. Otherwise, there are three cases for each overlay area O as the followings:

- There are no point in the overlay area O, then, we do nothing to adjust it.
- There are some points in the overlay area O, nevertheless, no points are on the edge of O. In this case, adjust these points will not effect the area of O. Therefore, we do not adjust these points, either.
- There are some points in the overlay area O and at least one point is on the edge of O. Then we could adjust the points on the edge of O to the other D_i it does not belong to originally, with which to reduce the area of overlay O.

When no point is on the edge of any overlay area, this optimization process is performed.

Algorithm 5 MINOVERLAY $(D_1, \cdots, D_N)$

```
1: if $(D_i \cap D_j) = NULL(i \neq j)$ then
2:     return $D_1 \cdots D_N$
3: end if
4: SELECTUNDISPOSED $(O) = D_i \cap D_j$
5: if no point in $O$ then
6:     do nothing
7: else if points are not on the edge of $O$ then
8:     return $D_1 \cdots D_N$
9: else if points are on the edge of $O$ then
10:    for each $p \in$ EDGE $(O)$ do
11:        if $p \in D_i$ then
12:            $D_i \leftarrow D_i - p$ ; $D_j \leftarrow D_j + p$
13:        else
14:            $D_j \leftarrow D_j - p$ ; $D_i \leftarrow D_i + p$
15:        end if
16:    end for
17: end if
18: return MINOVERLAY$(D_1, \cdots, D_N)$
```

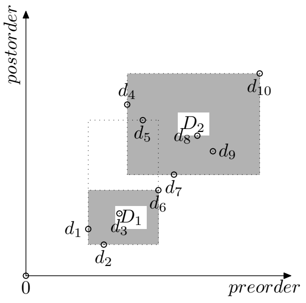


Figure 9: Modify Overlay

4.3 Indexing Partitioned Groups

We now introduce the data structure to index partitioning results D and A. There are much researches on indexing and querying spatial data. R-tree [17], R*-tree [7] and M-tree [10] are popular index mechanisms for indexing spatial data and spatial join [1, 6, 19, 25, 32]. R-tree and M-tree families have good search performance, which could be used in our data partition method. XML has inherent structural characters distinct from other spatial data, therefore, we adopt a more simple but effective method for partition.

We use *B^+-Tree* to index the groups of D, after spatial partitioning of D, into N groups, $D_1, D_2, \cdots D_N$. For each D_i, we index its corresponding A which may produce join results with D_i. They are A_{i_1}, A_{i_2}, A_{i_3} and A_{i_4}. We call this structure partitioned spatial structural index tree (*PSS-tree*). The structure of *PSS-tree* is shown in Figure 10. All data in *PSS-tree* are clustered.

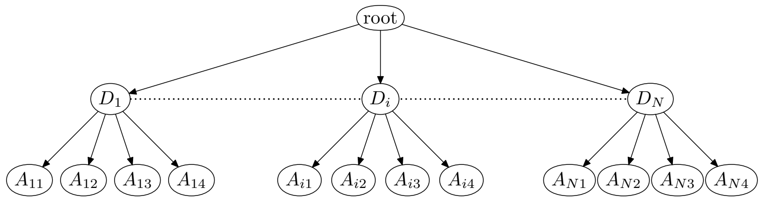


Figure 10: Data Structure of PSS-tree

4.4 Partitioned Spatial Structural Join Algorithms

Based on the SPATIAL-PARTITIONING method, spatial partitioning methods and *PSS-tree*, in this section, we propose two partitioned spatial structural join algorithms, *PSSJ* and *OPSSJ*.

Partitioned spatial structural join algorithm (PSSJ) first filters most ancestors and descendants which will not be join results. Then it uses basic SPATIAL-PARTITIONING method to partition D. Next we use a *PSS-tree* to index all partitioning groups of D and corresponding As. As to each group, we call SPATIAL-PARTITIONING method to perform the join operation. Now we give the PSSJ algorithm.

Algorithm 6 PSSJ (A, D, N)

```
1: FILTER $(A, D)$
2: SPATIAL-PARTITIONING $(D, N)$
3: for each $D_i \in D_1 \cdots D_N$ do
4:     PSSTree.insert($D_i, A_i$)
5: end for
6: for each $i \in 1 \cdots N$ do
7:     P-JOIN $(A_i, D_i)$
8: end for
```

The process of the optimized partitioned spatial structural join algorithm OPSSJ is similar to PSSJ. The only difference is that after partitioning D, OPSSJ calls two optimized methods to optimize the partitioning result. The optimized algorithm OPSSJ are given below:

Algorithm 7 OPSSJ (A, D, N)

```
1: FILTER $(A, D)$
2: SPATIAL-PARTITIONING $(D, N)$
3: MINMAX $(D_1 \cdots D_N)$
4: MINOVERLAY $D_1 \cdots D_N$
5: for each $D_i \in D_1 \cdots D_N$ do
6:     PSSTree.insert($D_i, A_i$)
7: end for
8: for each $i \in 1 \cdots N$ do
9:     P-JOIN $(A_i, D_i)$
10: end for
```

5 Performance Analysis

In this section, we present the experimental evaluation that yields a sense for the efficacy of our novel partitioned structural join algorithms. We start with a description of our experiment platform and test data set in Section 5.1. The following sections present a result evaluation and performance evaluation of our partitioned spatial structural join algorithm.

5.1 Experiment Setting

We ran experiments on a PC with Intel Pentium 2.0 GHz CPU, 512M RAM and 40G hard disk. The operating system is Microsoft Windows XP. We employ an OODBMS FISH [30] to store XML data. The testing programs were written in Forte *C++* and INADA conformed to ODMG [8] *C++* binding. We have implemented partitioned SpatialStructuralJoin (PSSJ) and its optimized algorithm(OPSSJ), StackTreeJoin (STJ) and *XR-Tree* algorithm (*XR-Tree*).

We utilize synthetic data for all our experiments in order to control the structural and consequent join characteristics of the XML documents. We use the IBM XML data generator [15] to generate XML document. The two DTDs adopted are shown in Figure 11. Different sized documents that scale from 20M to 100M are generated with each DTD.

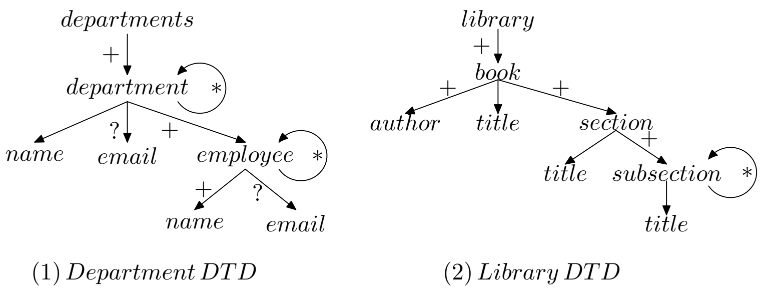


Figure 11: DTDs for Synthetic Data

QUERY	DESCRIPTION	DTD
Q_1	*department//employee*	`Department`
Q_2	*employee//name*	`Department`
Q_3	*section//subsection*	`Library`
Q_4	*section//title*	`Library`

Table 1: Query Set

Experiments were performed to study the comparative performance on our two partitioned spatial structural join algorithm. Table 1 shows the set of queries for all data set on two DTDs.

Table 2 presents the nested case of our query set. Q_1 is selected to test the case that both ancestor and descendant are highly nested. Q_2 is selected to test the case that ancestor is highly nested but descendant is less nested. Q_3 is selected to test the case that ancestor is less nested but descendant is highly nested. Q_4 is selected to test the case that both ancestor and descendant are less nested. The test query set contains all possible nested case of ancestor and descendant for thoroughly analyzing the performance of partitioned spatial structural join algorithms.

QUERY	ANC NESTED	DESC NESTED
Q_1	*highly*	*highly*
Q_2	*highly*	*less*
Q_3	*less*	*highly*
Q_4	*less*	*less*

Table 2: Query Nested Case

5.2 Varying Partitioning Number

The objective of this set of experiments is to test the query performance of PSSJ and OPSSJ with varying partitioning number. We adopt one 100M document for each DTD. The performance curves of testing varying partitioning number are given in Figure 12.

Figure 12(a) shows the performance of Q_1 with partitioning number varying from 10 to 60. Figure 12(b) shows the performance of Q_2 with partitioning number varying from 10 to 60. Figure 12(c) shows the performance of Q_3 with partitioning number varying from 20 to 120. Figure 12(d) shows the performance of Q_4 with partitioning number varying from 30 to 180.

It can be seen from Figure 12 that PSSJ algorithm and OPSSJ algorithm can get better performance along with the increase of the partitioning number, which demonstrates the effectiveness of our partitioning methods. And in each case, the optimized algorithm OPSSJ outperforms the basic PSSJ algorithm. We can also see that when the partitioning number gets to some value in each case, the amplitude of performance will be slow. This value is closely relative to the number and the nested case of ancestors and descendants. The more the number of ancestors and descendants, the more partitioning number is needed to get good performance.

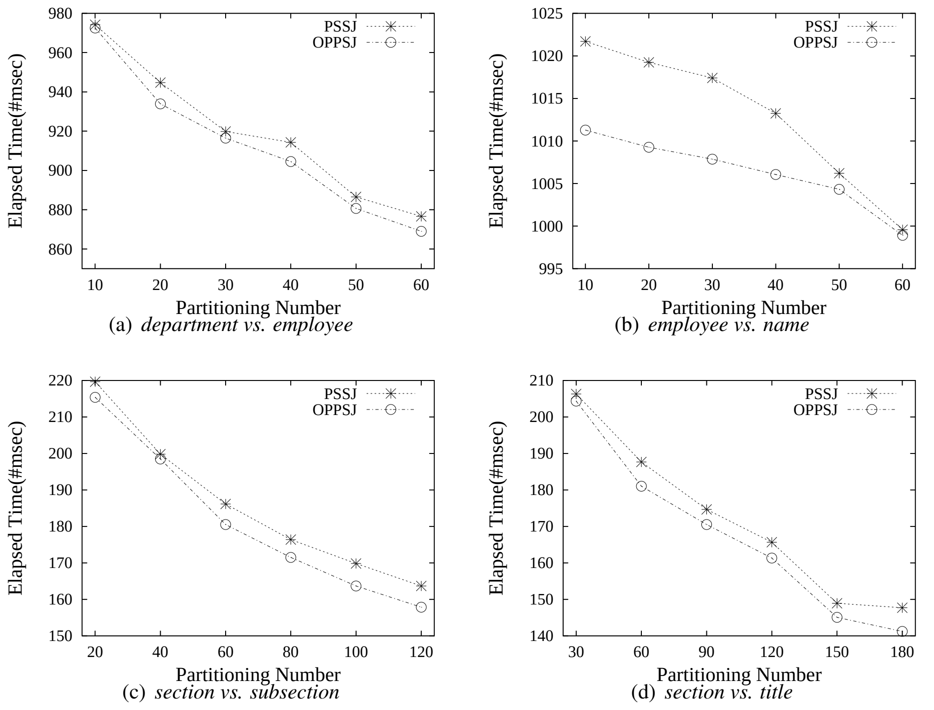


Figure 12: Varying Partitioning Number Test

5.3 Varying Join Selectivity Test

In the second group of experiments, we study the capabilities of various algorithms to skip elements under different join selectivity. We keep the document size 100M and partitioning number unchanged for each query in this test. We test the case when the ancestor join selectivity and descendant join selectivity are varying. For this purpose, we change the document elements with dummy elements so that the desired selectivity on ancestors or descendants can be obtained.

Figure 13 shows the performance of the four algorithms tested when varying join selectivity.

Figure 13(a) tests Q_1 and Figure 13(b) tests Q_2. The nested level of ancestors are high in both queries. We can clearly see from them that in this case, the join selectivity has great influence on *XR-Tree* algorithm, while has little influence on Stack-Tree join algorithm and our two partitioned spatial join algorithms. And in this case, *XR-Tree* has bad performance. Stack-Tree join algorithm has similar good performance as PSSJ and OPSSJ.

Figure 13(c) tests Q_3 and Figure 13(d) tests Q_4. The nested level of ancestors is low. We can see from (c)(d) that join selectivity has great influence on all tested algorithms. And in this case, *XR-Tree* is better than Stack-Tree join and has similar performance as our two partitioned spatial structural join algorithms.

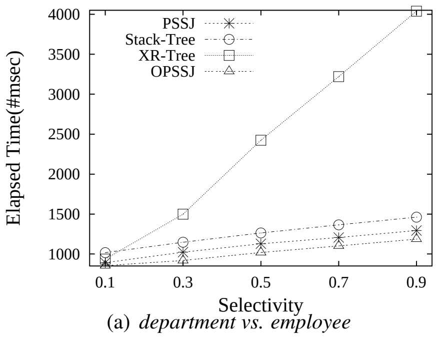
5.4 Sizeup Test

In the third set of experiments, we vary the document size to compare the performance of different structural join algorithms. We kept the join selectivity at 30%, we also give the fixed partitioning number for PSSJ and OPSSj algorithms. We vary the document size from 20M to 100M.

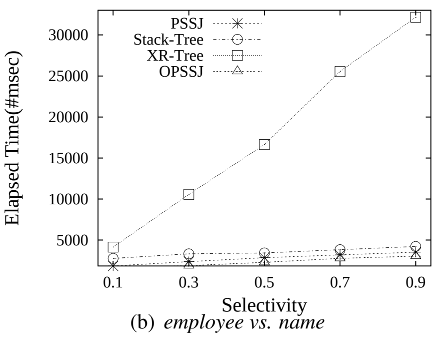
Figure 14 gives this group of performance test.

We can see that in all cases, our partitioned spatial structural join algorithms PSSJ and OPSSJ get better performance than *XR-Tree* and *XR-Tree* is better than Stack-Tree join algorithm. But 14(b) is an exception, when the nested level of ancestor is high and the nested level of descendant is low, *XR-Tree* get worse performance than Stack-Tree join algorithm.

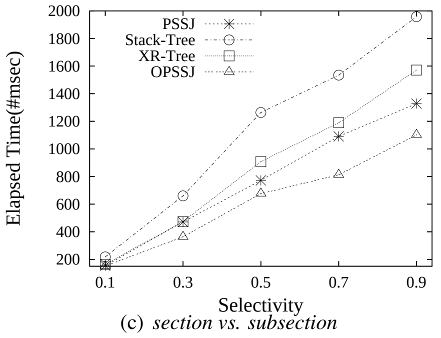
We can also observe from this figure that all four tested structural join algorithms have good sizeup performance. Along with the increase of the document size, the elapsed time of each algorithm increases



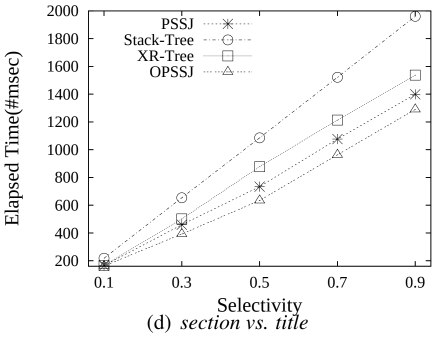
(a) *department vs. employee*



(b) *employee vs. name*

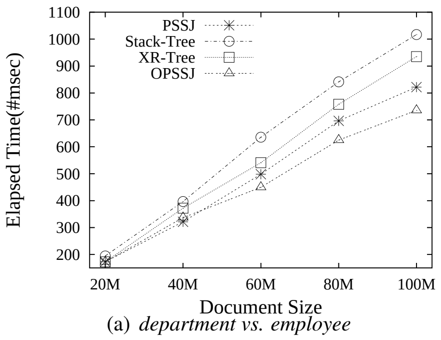


(c) *section vs. subsection*

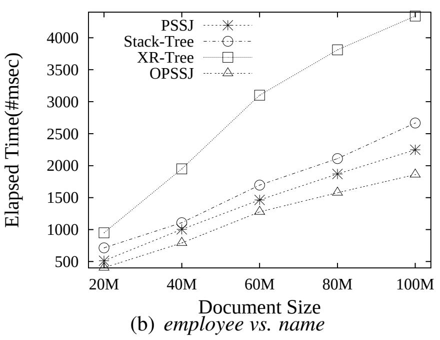


(d) *section vs. title*

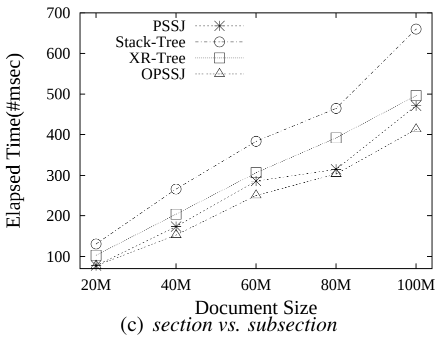
Figure 13: Varying Join Selectivity



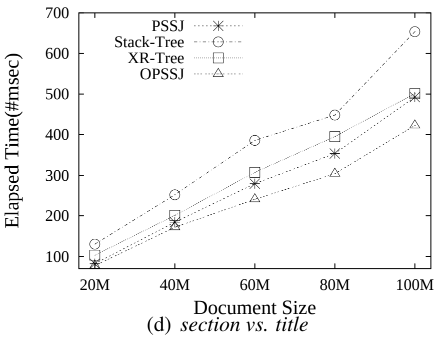
(a) *department vs. employee*



(b) *employee vs. name*



(c) *section vs. subsection*



(d) *section vs. title*

Figure 14: Sizeup Test

well-proportioned.

5.5 Partitioning vs Workload

In this subsection, we analyze the impact of partitioning on the workload of structural join.

Table 3 shows the workload of the sample XML document in Figure 1 under different number of partitions.

Table 3: Workload of A//D in the sample XML document

# OF PARTITIONS	A_{output}	A_{join}	WORKLOAD
1	1	6×10	60
2	2	2×5	25
	1	3×5	
	3	1×3	
3	2	0×4	12
	1	3×3	
	3	1×3	
4	2	0×2	12
	2	0×2	
	1	3×3	

A_{output} is the part which could directly output join results without join operation. Thus we omit the workload of this part of A. A_{join} is the part which needs structural join with corresponding partition D, we simply compute the workload as $n_a \times n_d$ where n_a is the number of a.

As shown in Table 3, along with the increasing number of partitions, the total number of a that falls in the A_{output} increases, the number of a which takes part in the join decreases correspondingly. Thus the total workload of A_{join} decreases. For example, when the number of partitions is 1, the join workload is 60. When the number of partitions is 2, the join workload turns to 25. When the number of partitions is 3, the join workload decreases to 12, but when the number of partitions is 4, the join workload is still 12 because the number of a and d participating in the join is changeless. So in this case, we need not further partitioning operation.

Figures 15 and 16 give the impact of partitioning on the workload of structural joins of *section//title* and *department//employee*. From the tables, we can see that with the increasing of the number of partitions the workload of structural join is decreasing while the number of nodes in the partition which can be used to output the join results is increasing.

# OF PARTITIONS	A_{output}	WORKLOAD
10	0	657009
20	0	376814
30	0	283415
40	2	231936
50	6	197166
60	13	167544
70	21	147767
80	29	131672

Figure 15: Workload of section//title

# OF PARTITIONS	A_{output}	WORKLOAD
10	0	657009
20	0	376814
30	0	283415
40	2	231936
50	6	197166
60	13	167544
70	21	147767
80	29	131672

Figure 16: Workload of department//employee

5.6 Maintenance Overhead Test

In the last experiment, we test the maintenance overhead of *XR-Tree* and our basic partitioned spatial structural join algorithm PSSJ. We mainly test the elapsed time of constructing index structures. Figure 17 gives the maintenance cost with documents of different size.

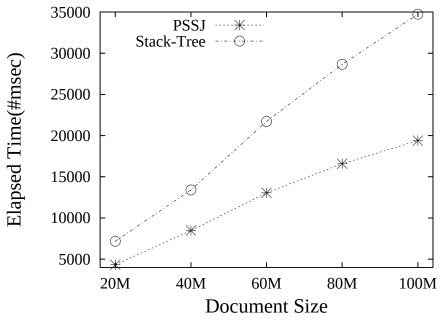


Figure 17: Maintenance Cost(*department vs. employee*)

We only take Q_1 for example because the other performance curves have the similar trend. Observe from this figure, we can see that the maintenance cost of PSSJ is always less than that of *XR-Tree*. With the increase of the document size, this trend is more obvious.

6 Conclusions

In this paper, we first extend the relationships between nodes to the relationships between partitions and get some observations about the relationships between partitions. We then introduce a partition-based method *P-Join* [26], for structural join between ancestors and descendants based on the properties derived from these observations. Moreover we present an enhanced partitioned-based structural join algorithm and two optimized methods. In order to store the partitioning results, we design a simple but efficient index structure called *PSS-tree*. Extensive experiments show that our approaches outperform previous methods and also we could get low maintenance cost.

Acknowledgement

This work is partially supported by the CUHK strategic grant (numbered 4410001).

References

[1] L. Arge, M. d. Berg, H. J. Haverkort, and K. Yi. The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, Paris, France, 2004.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of Int. Conf. on Data Engineering (ICDE) '02*, San Jose, California, USA, 2002.

[3] E. Babb. Implementing a Relational Database by Means of Specialized Hardware. *ACM Transactions on Database Systems (TODS)*. 4(1), 1-29, 1979.

[4] A. Berglund, S. Baog, D. Chamberlin, et al. XML Path Languages (XPath). W3C Working Draft. Technical Report WD-xpath20-20011220, http://www.w3.org/TR/WD-xpath20-20011220, 2001.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, pages 310-321, Madison, Wisconsin, 2002.

[6] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-Trees. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, pages 237-246, Washington, DC, USA, 1993.

[7] B, Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, pages 322-331, Atlantic City, NJ, 1990.

[8] The Object Database Standard: ODMG 3.0. *Morgan Kaufmann Publishers, Inc*, San Francisco, California, 2000.

[9] D. Chamberlin, J. Clark, D. Florescu, et al. XQuery 1.0: An XML Query Language. W3C working draft. Technical Report WD-xquery-20010607, 2001.

[10] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proc. of the 23th Int'l Conference on Very Large Data Bases (VLDB)*, pages 426-435, Athens, Greece, 1997.

[11] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proc. of Third Int. Workshop WebDB*, 2000.

[12] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. of the 28th Int'l Conference on Very Large Data Bases (VLDB)*, pages 263-274, Hong Kong, China, 2002.

[13] P. F. Dietz. Maintaining order in a linked list. In *Proc. of the Fourteenth Annual ACM Symposium on Theory of Computing*, 1982.

[14] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, XML-QL: A query language for XML. http://www.w3.org/TR/NOTE-xml-ql., 1998.

[15] A. Diaz and D. Lovell. XML generator. http://www.alphaworks.ibm.com/tech/xml generator, Sept. 1999.

[16] T. Grust. Accelerating XPath Location Steps. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, Madison, Wisconsin, USA, 2002.

[17] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, Boston, Massachusetts, 1984.

[18] T. Grust, M. v. Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. In *Proc. of the 29th Int'l Conference on Very Large Data Bases (VLDB)*, pages 524-525, Berlin, Germany, 2003.

[19] Y.-W. Huang and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimization. In *Proc. of the 23th Int'l Conference on Very Large Data Bases (VLDB)*, pages 396-405, Athens, Greece, 1997.

[20] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient stuctural join. In *Proc. of Int. Conf. on Data Engineering (ICDE) '03*, Bangalore, India, 2003.

[21] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proc. of the 29th Int'l Conference on Very Large Data Bases (VLDB)*, pages 273-284, Berlin, Germany, 2003.

[22] Q. Li and B. Moon. Indexing and querying XML Data for regular path expressions. In *Proc. of the 27th Int'l Conference on Very Large Data Bases (VLDB)*, pages 351-360, Roma, Italy, 2001.

[23] Q. Li and B. Moon. Partition Based Path Join Algorithms for XML Data. In *Proc. of the Database and Expert Systems Applications (DEXA)*, Prague, Czech Republic, 2003.

[24] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A Queriable Compression for XML Data. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, San Diego, California, USA, 2003.

[25] G. Proietti and C. Faloutsos. Analysis of Range Queries and Self-Spatial Join Queries on Real Region Datasets Stored Using an R-Tree. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12(5): 751-762, 2000.

[26] N. Tang, J. X. Yu, K.-F. Wong, K. Lu and J. Li. Accelerating XML Structural Join By Partitioning. In *Proc. of the Database and Expert Systems Applications (DEXA)*, Copenhagen, Denmark, 2005.

[27] P. Valduriez and G. Gardarin. Join and Semi-Join Algorithms for a Multiprocessor Database Machine. *ACM Transactions on Database Systems*, 9(1): 133-161, 1984.

[28] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree Coding and Efficient Processing of Containment Joins. In *Proc. of Int. Conf. on Data Engineering (ICDE) '03*, Bangalore, India, 2003.

[29] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proc. of Int. Conf. on Data Engineering (ICDE) '03*, Bangalore, India, 2003.

[30] G. Yu, K. Kaneko, G. Bai, and A. Makinouchi. Transaction Management for a Distributed Store System WAKASHI Design, Implementation and Performance. In *Proc. of Int. Conf. on Data Engineering (ICDE) '96*, New Orleans, Louisiana, 1996.

[31] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data (SIGMOD)*, Santa Barbara, CA, USA, 2001.

[32] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate Similarity Retrieval with M-Trees. *VLDB Journal*, 7(4): 275-293, 1998.